PRECIP: Towards Practical and Retrofittable Confidential Information Protection

XiaoFeng Wang[†], Zhuowei Li[†], Ninghui Li[‡] and Jong Youl Choi[†]
[†]Indiana University at Bloomington. [‡]Purdue University.

Abstract

A grand challenge in information protection is how to preserve the confidentiality of sensitive information under spyware surveillance. This problem has not been well addressed by the existing access-control mechanisms which cannot prevent the spyware already in a system from monitoring an authorized party's interactions with sensitive data. Our answer to this challenge is PRECIP, a new security policy model which takes a first step towards practical and retrofittable confidential information protection. This model is designed to offer efficient online protection for commercial applications and operating systems. It intends to be retrofitted to these applications and systems without modifying their code. To this end, PRECIP addresses several practical issues critical to containing spyware surveillance, which however are not well handled by the previous work in access control and information-flow security. Examples include the models for human input devices such as keyboard whose sensitivity level must be dynamically determined, other shared resources such as clipboard and screen which must be accessed by different processes, and the multitasked processes which work on public and sensitive data concurrently. We applied PRECIP to Windows XP to protect the applications for editing or viewing sensitive documents and browsing sensitive websites. We demonstrate that our implementation works effectively against a wide spectrum of spyware, including keyloggers, screen grabbers and file stealers. We also evaluated the overheads of our technique, which are shown to be very small.

1 Introduction

Although protecting confidential information has long been recognized as one of the most important security problems, never before has the demand for its practical solutions been so imperative. A recent study by Webroot revealed that about 89% of computers it scanned were infected with spyware, with an average of 30 instances per machine [1]. This threat can be mitigated by the techniques which aim at preventing spyware from being installed [18] or detecting and disinfecting spyware-riddled hosts [35, 31]. However, reliance on these techniques as the only defense is risky, as evasion of them leaves confidential information completely unprotected. Therefore, it is crucial to enable a host to *contain* spyware surveillance, preventing data from being stolen even after attackers manage to breach other layers of defense. Unfortunately, this cannot be achieved by the access control mechanisms running in current commercial systems: for example, though mainstream word processing software such as `Microsoft Word` offers password and encryption protection to sensitive files, a keylogger can easily get around such defense by recording the password used by an authorized party to access these files. Fundamentally, these mechanisms are designed to regulate access to resources, not to control propagation of the information released from the resources [6, 10, 58].

Complementary to these mechanisms are the technologies for *information flow security* [33, 47]. The idea is to track and manage sensitive data to prevent them from flowing into unauthorized parties. Research in this area started with the famous Bell-LaPadula (BLP) model [21, 39]. The BLP model is designed to regulate the information flows between *subjects* (e.g., processes) and *objects* (e.g., files) with different sensitivity levels. Informally, the model forbids a subject from reading objects with higher sensitivity levels, or writing objects with lower sensitivity levels. However, these properties can be too restrictive for many commercial systems, in which most applications are multitasked and expected to work concurrently on the objects with various sensitivity levels. Moreover, BLP does not model the resources and input devices shared between sensitive subjects and public subjects (e.g., clipboard, screen and keyboard), which are widely used by spyware to gather sensitive information from the user. More recent research on information flow security focuses on tracking and controlling information flows within a program [47, 56, 44]. Many proposals require modifying source code to enhance it with information-flow policies. However, the source code of commodity software is usually not publicly available. Alternatively, some approaches instrument the binary code of

an executable [51, 48] or dynamically track its execution using virtual machines or debugging tools [29, 43]. A problem is that instruction-level tracing incurs significant performance penalties, which at least triple a program's running time [51]. This makes these approaches more suitable for offline analysis than online protection of information [29, 43, 51].

We believe that information-flow based approach is promising to achieve confidentiality protection, but its implementation in practical systems can be hard unless the technical barriers discussed above have been effectively addressed. Specifically, an information-flow model must fully reflect the security-related properties of operating systems and applications, and be efficient enough for using online. It is also expected to be retrofitted to legacy systems without touching their code and largely preserve the way users interact with them. As a first step towards these objectives, we propose PRECIP, a new *confidentiality model* that aims at practical and retrofittable confidential information protection. PRECIP controls the sensitive information flows among subjects and objects through tracking individual subjects' outputs which are *dependent* on their sensitive inputs: for example, data transferred from a word processor to clipboard are deemed dependent upon the program's inputs from the document being edited. Our model prevents sensitive information from getting into untrusted subjects and dynamically identifies the new objects which receive sensitive data. It also explicitly describes shared resources and user input devices such as screen, shared memory and keyboard, and specifies the action which changes their sensitivity levels. For example, whenever a user starts running a word processor to edit a sensitive document, our approach automatically tracks its input flows from keyboard and its output flows to screen, shared memory (e.g., clipboard) and temporary files, and protects them from being accessed by untrusted subjects through the activities such as intercepting keystrokes or taking a snapshot of the screen. PRECIP is designed for online operation and can be retrofitted to legacy operating systems and applications without changing their code. In our research, we implemented our model to protect some Windows applications, which demonstrates its effectiveness in preventing information leakage during common data operations such as viewing or editing documents, and browsing websites.

The main contributions of the paper are described as follows:

- **Effective and efficient confidentiality protection**. PRECIP can model user inputs, shared resources and multitasked applications. It is designed to be used online. Our implementation of the model under Windows XP was demonstrated to work effectively against a large spectrum of spyware, including different types of keyloggers, screen grabbers and Trojans. Our experimental study also shows that PRECIP introduces only small overheads to the operating system and applications it protects.

- **Retrofit to legacy systems and ease of use**. The security policies of PRECIP can be enforced by system-call interposition, which can be achieved using the toolkits such as Wrappers [36], and user-land programs such as hooks for Windows message-handling mechanism and applications' add-ons. This avoids modifying the source and binary code of operating systems and applications, and also makes our approach easy to switch on and off.

- **New techniques to trace and control sensitive information flows**. We propose simple but effective dependency rules to determine the dependency relations between inputs and outputs of some Windows applications, including word processing software and Web browsers. We also designed a novel framework to protect sensitive message traffic within Windows message-handling mechanism, which prevents untrusted hooks from intercepting sensitive messages but still allows them to work properly on public messages.

Developing practical information-flow based confidentiality protection is well recognized to be a challenging task. PRECIP takes a first step towards this objective. Inevitably, it contains limitations that need further investigation. First, our approach generates dependency rules empirically, based upon application-specific knowledge. Second, those rules are designed to be general to a category of applications without modifying their code. However, this property also makes them less precise: there is no guarantee that all the sensitive outputs of an application will be captured by the rules, which may potentially let sensitive data slip into an unauthorized party. Third, enforcement of PRECIP policies and dependency rules are also achieved in an empirical manner, relying on a set of control programs specific to applications. These problems could be addressed in the future research by techniques for automatic analysis of applications and extraction of their dependency rules, and a general framework for enforcing these rules and other policies. Another problem is that PRECIP interferes with the operation of a legitimate application, which could undermine the functionality of the applicatoin under some circumstances, for example, when it is processing both sensitive and public data. Finally, PRECIP does not model integrity and relies on additional integrity protection to ensure that it is not bypassed. While our prototype implementation includes a basic integrity protector, its protection is not comprehensive, and a practical, comprehensive integrity model will certainly complement our work.

The rest of the paper is organized as follows: Section 2 presents our model; Section 3 describes a design which applies the model to protect some data operations under Windows XP; Section 4 gives the details of our proof-of-concept implementation; Section 5 reports our experimental studies on the implementation; Section 6 discusses the limitations of our approach; Section 7 surveys the previous work related to PRECIP, and Section 8 concludes the paper and envisions the future work.

2 The Model

The PRECIP model for confidentiality protection is motivated by the following issues in modern commercial operating systems (such as Microsoft Windows), which are not effectively handled in existing confidentiality protection Mandatory Access Control (MAC) models such as BLP or non-interference [21, 30].

User Input Objects Existing MAC models label information based on their source. For example, a system may have some files labeled as "high" and some labeled as "low", and information derived from "high" files is considered "high". While this approach works for information that already exists in the system, it does not work for information newly generated from user-input objects, such as the keyboard. Because the keyboard is shared by applications running at different levels, the events they generated are also at different levels.

Consider the scenario that one downloads and installs an input method program for inputting in a foreign language. The program needs to receive keyboard events. However, since the program could potentially be a keylogger and cannot be fully trusted, we would want to prevent the program from receiving keyboard events while one is typing, for example, passwords for logging into a bank's website. In this scenario, given a keyboard event, one has to *predict* its sensitivity, based on which application these events will be eventually sent to. For example, if the events are sent to a web browser that is accessing the website of the user's bank, then these events should be considered sensitive, because they may include passwords for logging into the user's account. On the other hand, if one is typing while posting on a public forum, then the keyboard should not be considered to be sensitive.

Therefore, if one assigns a label to the keyboard, then this label must be dynamic. It depends upon whether the keyboard will be used as input for high objects in the future. This is different from existing models, where the label of an object is derived from the past history of how this information is derived. When labeling a new keyboard event, one has to *predict* how it will be used.

Other Shared Objects Many objects other than user-input objects are also shared by multiple subjects. Two examples are the screen and the clipboard. These objects are critical in order to have a functional system, and they need protection. One class of spyware, known as screen grabber, extracts information from the screen. These shared objects will also have dynamic labels. While some of the objects such as the screen can drop its sensitivity level by itself after an event is terminated, other shared objects may require an explicit "cleaning" operation to remove the content to enable its sensitivity level to drop.

Multi-tasked subjects Operating system kernels maintain protection boundary at the level of processes. Access within a process's address space is generally unmediated. However, in modern operating systems, one process may contain multiple threads that serve different tasks, often at different sensitivity levels. For example, a single `Word` process may contain multiple threads that are used to edit both sensitive files and non-sensitive files. Similarly, one browser may be used to browse websites of different sensitivity.

In the BLP model, one has to either forbid such situation or declare the subject as "trusted", so that the subject can write arbitrarily. A "trusted" subject in BLP is trusted to perform declassification correctly. However, this treatment does not work for modern OS such as Windows, in which most applications are multithreaded and many of them do not have declassification capability. For example, there is no obstruction in `Word` to copying data from a sensitive document and pasting them to a public document.

2.1 Basic Concepts

We now introduce the basic concepts in PRECIP. *Objects* are repositories for information. An object can be either local or remote (across the network). Examples include files, buffers, keyboard, screen, websites (remote), etc. A *user-input object (UIO)* is an object through which subjects receive user inputs. Examples of UIO include keyboard and mouse. *A subject* is an information processing unit which operates on objects. We treat each process as a subject.

A channel connects a subject to subject, a subject to an object, or an object to a subject. For example, opening a file for read can be viewed as the operation to open a channel from the file to the subject. A network connection can be viewed as a two-way channel between a local subject (process) and a remote host (modeled as an object). *A path* consists of multiple channels that are connected by subjects. That is, a path has the following form: (u_0, u_1), (u_1, u_2), $\ldots$, (u_k, u_{k+1}), where each (u_i, u_{i+1}) is a channel, u_0 and u_{k+1} are either subjects or objects, and $u_1, \ldots, u_k$ are subjects. For example, under Microsoft Windows, multiple subjects can hook on the delivery of keyboard events. They form a path through with a keyboard event will be delivered.

Information flows from a subject/object to another subject/object through channels by *messages*. Our notion of message is general. Any communication is viewed as

the passing of a message. Examples of messages include keyboard events, mouse events, data transferred through a file writing system call, data transferred through an interprocess communication call, etc. Each message emitted by a subject may *depend upon* a set of input messages received by the subject. The concept of dependency captures the intuition that information may flow through a subject. This dependency relation is assumed to be an input to the model. Our model does not force a particular way of tracing the dependency and allows the use of any approach for determining the dependency. In BLP, there are two kinds of subjects. An untrusted subject must satisfy the *-property (i.e., no write-down). This can be interpreted as assuming each output message depends upon *all* previous input messages. A trusted subject is not restricted by the *-property restriction. This can be interpreted as assuming each output message emitted by the subject depends upon *no* input message. The BLP approach is limited by the fact that it treats subjects as blackboxes, and has only the two extreme cases of dependency. It thus cannot handle multitasked processes. PRECIP handles multitasked processes by allowing finergrained dependency rules to be specified and used. One example dependency rule we use in our implementation is as follows. Under Windows, when one opens multiple files in one `Word` process, the subject corresponding to the process receives messages from all files. However, when the subject writes, the output message depends only upon the file being edited by the user (Section 3.3).

Every object has *a sensitivity level*, which can be an element in a lattice. In our implementation, the level is either high or low. We thus focus our description of PRECIP to this case, though our model can be extended to the more general case where the set of all sensitivity levels forms a lattice. An object is said to be *sensitive* if its sensitivity level is high.

An object's sensitivity level may change. It may raise from low to high if it receives a sensitive message. It may also drop from high to low if one performs a "clean" operation on it. For example, when one overwrites the content of a clipboard, the system first "cleans" the clipboard, enabling its sensitivity level to become low. Cleaning ensures that all future messages will not depend upon previous incoming messages.

An object may be *remote* or *local*. A remote object is not under the control of the security system. Technically, a remote object has a sensitivity level (which can be either high or low) that cannot be changed by the reference monitor. Sensitive information should not be sent to remote objects with low sensitivity levels, because one cannot control who can read those objects. For example, a remote website should be viewed as a remote object. On the other hand, a file on local hard drive can be viewed as local, because the protection system can limit which subjects can read the file. A file on an USB thumb drive should be viewed as remote, because the drive may be taken away and be used outside the protection system. Note that an USB thumb drive can still receive sensitive information if it is labeled as high.

A subject may be *trusted* or *untrusted*. A subject is untrusted if one has no confidence in controlling information flow through the subject by controlling its output channels mediated by the reference monitor. This would be the case if the program running in the subject is downloaded from an unknown source and may use covert channels to leak information. Sensitive information should not be sent to untrusted subjects. Note that our notion of trusted subjects is different from that in, for example, the BLP model. In BLP, trusted subjects are assumed to be able to correctly declassify information. Our assumption is weaker. In PRECIP, trusted subjects are assumed to behave in a way such that one can model information flow through them using the dependency rules. In essence, this implies that trusted subjects are assumed not to use channels not controlled by the reference monitor (e.g., covert channel) to leak information.

2.2 Security Objective

Intuitively the security objective is that any information flow that violates the confidentiality goal is not allowed. For example, spyware such as keyloggers should not be able to steal passwords. When we have just two sensitive levels: sensitive (i.e., "high") and public (i.e., "low"), the security objective can be stated as: *sensitive information should not be leaked*.

By leaked, we mean that information is delivered either to an untrusted subject or to a remote object with sensitivity level "low". As an untrusted subject may contain covert channels to leak information, delivering information to an untrusted subject should be considered leaking. As we cannot control who read a remote object with "low" sensitivity, delivering information there is also considered leaking.

By sensitive information, we refer to the information that is derived from the information in a sensitive object. There are two cases. One is a message that depends upon (possibly transitively) a message that comes out of a sensitive object. The other is a message that depends upon (possibly transitively) a message coming out from a UIO that is generating sensitive user inputs.

2.3 Policies Achieving the Objective

We now describe a collection of policies that together achieve the security objective identified above. These policies are divided into those that trace sensitivity levels of messages and objects and those controlling the delivery of messages.

A. Tracing sensitivity levels of messages and objects:

- Each message has a sensitivity level. When a message is generated, it is set to high if one of the following is

true: (1) the message is emitted from a sensitive object, (2) the message is emitted from a subject, and among the set of messages that the message depends on, at least one message is sensitive.

- Whenever an object receives a sensitive message, the object's sensitivity level is set to high.
- A UIO object is set to high if and only if there is a path connecting an UIO to a sensitive object.
- When an object is cleaned, then the object's sensitivity is set to low.

These policies trace sensitivity levels of objects and messages. Note that in our model, subjects do not have sensitivity levels; only objects and messages do. (A subject has a trust level; it can be trusted or untrusted.) This is because a subject may be operating on data of multiple sensitivity levels. Also note that the sensitivity level of a message generated by a subject is determined by the sensitivity levels of the messages it depends on.

B. Controlling: Message delivery is controlled according to the follow two policies: (1) If a sensitive message is being sent to an untrusted subject or remote object with lower sensitivity level, then the message is blocked. (2) If a sensitive message is being sent to a local object with lower sensitivity level, we have two choices: block the message, or deliver the message and mark the object as sensitive.

We now show that these policies achieve the security objective in Section 2.2, under two assumptions. First, whenever a message is outputted by an UIO, if there is no path connecting the UIO to an object, then the message (or other messages depending on it) will never be delivered to the object. Second, the cleaning operation is performed correctly, that is, no message sent after the cleaning depends upon any message received before the cleaning. Under the above two assumptions, a straightforward induction shows that the two kinds sensitive information identified in the security objective (Section 2.2) will be both correctly labeled as sensitive by the tracing policies above. The message delivery control then ensures that sensitive information is not leaked.

Additional policies can be introduced in an actual implementation, so long as the security objective are met. For example, when implementing the PRECIP model, we introduce a policy that blocks information delivered to a trusted subject, when the subject is communicating with a remote object labeled as "low" and we want to avoid blocking that communication.

3 Protecting Confidential Information in Windows XP

In this section, we present a design which applies the PRECIP model to Windows XP. Our objective is to protect the sensitive information related to two common data operations: viewing or editing sensitive documents and browsing sensitive websites. We first illustrate the general idea behind our design and then elaborate on its components.

3.1 Overview

Our design includes four major components: a *classifier*, a *tracer*, a *controller* and an *integrity protector*. The classifier identifies sensitive documents and trusted applications. The tracer detects the sensitive outputs of a trusted application and labels these outputs according to the tracing policies described in Section 2.3. The control policies of PRECIP (Section 2.3) are enforced by the controller through a kernel driver that regulates system calls and a userland hooking management mechanism that controls the information flow within Windows message-handling mechanism. The integrity of all these components is protected by the integrity protector. Figure 1 illustrates the design.

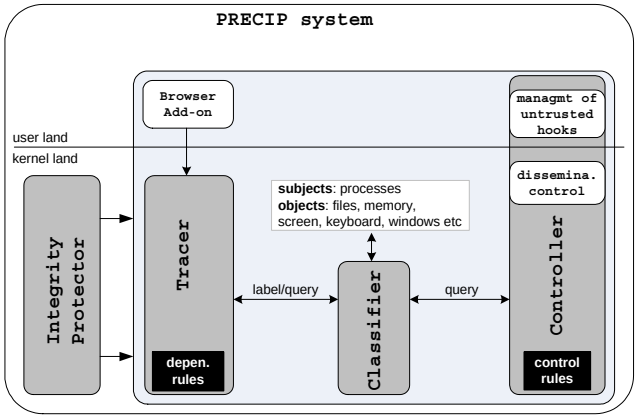

Figure 1. PRECIP on Windows XP.

Example. The general idea of our approach can be described with the following example. After the PRECIP mechanism has been installed, the *classifier* automatically labels trusted executables according to the presence of dependency rules, and identifies sensitive files. Suppose `Microsoft Word` has been labeled as a trusted application. Whenever it is launched, the classifier marks its process as a trusted subject. Once the process opens a sensitive file, the *tracer* employs dependency rules to identify the information emitted from the process which is deemed dependent upon the sensitive inputs from the file and raises the sensitivity levels of the objects receiving such information. These objects include screen on which an editing window displays the contents of the file, clipboard that contains the data copied from the file and temporary files which `Word` creates for the file. In addition, the tracer also sets keyboard and mouse as sensitive objects when the editing window

for the sensitive file is receiving data from these peripherals. The *controller* blocks the attempts from untrusted processes to read sensitive information, and cleans shared resources containing sensitive data. For example, when a sensitive editing window awaits keystroke inputs, the controller uses the hooking management mechanism to deliver keystroke messages without going through untrusted hooks; once the sensitive window stops receiving inputs, the controller purges clipboard and the key status of keyboard to lower down their sensitivity levels. At the meantime, the user's interactions with the editing window for a public document are allowed to proceed as normal. The *integrity protector* protects the `Word` process against the untrusted processes that attempt to interfere with its operations or instrument its executables.

Adversary model. We assume that no spyware is inside the kernel at the time our mechanism is installed. Our implementation is mainly at the kernel level and therefore cannot protect sensitive information if spyware is already inside the kernel. However, our integrity protector can *prevent* untrusted code from being loaded into the kernel through system calls (Section 3.5).

PRECIP is not designed for preventing exploit of software vulnerabilities, such as control-flow hijacking through buffer overrun. We rely on existing approaches such as the techniques for memory protection [26, 22, 53] to defend trusted applications and drivers against such attacks.

3.2 Classification and Labeling

The classifier is designed for labeling executables with trust levels and other objects with sensitivity levels.

Trust levels. In the PRECIP model, a subject is deemed trusted if we have the knowledge about its input/output dependency relations. Such knowledge is described in our design as a set of dependency rules, which we discuss in Section 3.3. Once installed, the PRECIP mechanism runs the classifier to locate all the executables of which dependency rules can be found from a database. These executables are labeled as trusted subjects after their integrity has been verified[1]. An authorized party (such as a system administrator) is also allowed to add new dependency rules[2] or modify existing rules through a security administrative tool.

The classifier labels an executable using its *file stream* [9]. Under an NTFS file system, a file can have multiple streams of data, of which the main stream is directly retrievable through the file name, and other streams are accessed through both the file name and stream names. For example, the main stream of a file "`myfile.ext`" is associated with its file name while its other streams are opened, read and written through the names in the format "`myfile.ext:stream-name`". Our classifier creates a stream "`trust-level`" for every executable file to label its trust level. This stream is protected against unauthorized modification by the integrity protector. A process is considered to be trusted if all the executables it is generated from are trusted. We label the trust level of a process using an unused byte in the KPROCESS structure of that process. The byte locates at offset `0x33`. The KPROCESS structure is inside the Windows kernel, and therefore cannot be accessed by userland code.

Sensitivity levels. The classifier also creates for every nonexecutable file a stream "`sensitivity-level`". Similar to the `trust-level` stream, this stream can only be changed by the classifier. File labeling is a process that needs user intervention. However, automatic tools can be used there to reduce the burden of manually annotating individual files. An approach is inferring the sensitivity level of a file according to its discretionary access control (DAC) information. Windows XP allows the user to set access polices of her files, including the actions such as "modify", "read", "write" and "execute" other users are permitted to take on the files. In addition, it can also encrypt the files on the user's request. Our classifier can automatically label the files which are encrypted or only accessible by its owner and the system administrator as sensitive objects. It also utilizes heuristics to identify sensitive files. For example, a ".pst" file is deemed sensitive. The user is also allowed to change a file's sensitive level and even designate a label to a newly created file through an interface provided by the classifier. The interface offers a list of windows through which the user can determine the sensitivity level of a yet-to-save file.

Besides files, other types of objects also have sensitivity levels. For example, windows, screen, keyboard and clipboard can be labeled with various sensitivity levels according to tracing policies (Section 2.3). These sensitivity levels are recorded in an array shared by the classifier, the tracer and the controller. Remote resources such as websites also have sensitivity levels. Our mechanism maintains a list of sensitive Internet domain names such as those of banks and other financial institutions and a set of sensitive IP prefixes.

3.3 Sensitive Information Tracing

The tracer first detects information flows coming out of sensitive objects, and then employs a set of dependency rules to identify the sensitive outputs of a process that receives these flows. In our research, we implemented the tracer in a kernel driver to intercept system calls, and userland programs to help identify sensitive information. Such userland programs include an add-on for enforcing the de-

[1] For example, we can check their hash fingerprints against those of known legitimate programs.

[2] Those rules can be constructed by analyzing an application or a category of applications.

pendency rule for web browser and a hooking management mechanism for tracking the information within Windows message-handling mechanism. The latter is described in Section 3.4 as it is also a part of the controller.

Identifying and tracing sensitive information. The tracer monitors the operations performed by processes on sensitive objects to detect sensitive information flows. Examples of such operations include reading from a sensitive file and navigating to a sensitive URL. In addition, the tracer automatically marks input peripherals such as keyboard and mouse as sensitive objects whenever *input focus* [3] is set on a window processing sensitive information, which indicates it awaiting inputs from these devices. In this case, the information flows produced by these devices such as keystrokes and mouse movements are also deemed sensitive.

A dependency rule for editing or viewing applications. Many Windows-based editing/viewing applications do not support multiple processes: for example, `Word` maintains a single process to serve all documents. This prevents us from utilizing process boundary to separate information in these applications without altering their code. Therefore, our current design tracks sensitive outputs of these applications using a heuristic dependency rule based on an intuition described below.

Editing or viewing applications aim at providing the user with interfaces and tools to process her documents. Therefore, most of their outputs are directly triggered by human inputs. Under Windows XP, these applications use a graphic user interface called *window* to visualize a document to the user and let her edit its contents. We call such a window the *hosting window* of the document. A top-level window in Windows is called an *application window*, which has no owner window and appears on Windows taskbar. A window can further spawn child windows to handle various tasks such as interactions with file system and the user. The contents of a document are displayed by an application window in some applications, such as `Word`, and by its child window in others[3], such as `Excel`. A window is treated as an object by PRECIP. Its sensitivity level becomes high when it hosts a sensitive document or it is a child window of a sensitive window. A process of a Windows-based editing/viewing application can create multiple windows, each hosting a document. However, at any moment, only one window has the input focus on it for receiving user inputs. Such a window is called *active window*. From that window, we can identify which document the user is working on and then correlate the outputs of the application with it[4]. This observation is described by Rule 1 in Table 1.

Rule 1 can serve as a default rule for an editing/viewing application. In Section 5.1, we demonstrate that this rule works effectively on the commodity software such as `Microsoft Office`, `Adobe Reader` and `Notepad`. An important issue here is that our dependency rules are only meant to be applied to *legitimate software*, as tracing information flows of malware, even on the instruction level, is always subject to evasion by a cunning adversary. To tackle untrusted subjects, PRECIP adopts a pragmatic strategy which prevents sensitive information from getting into them in the first place.

To enforce Rule 1, we need to identify a sensitive window from the sensitive inputs generated by the operations such as opening and reading a sensitive file. For many Windows-based editing/viewing applications, this can be achieved through a simple technique described as follows. To allow the user to work on a sensitive file, such an application must display the contents of the file in an active window. Such a window can be created after the file is opened and read, or before these operations. In either case, the application will immediately update the window associated with the file. Our approach tracks the system calls following the calls for opening and reading a sensitive file. If a call for creating a new window (`NtUserCreateWindowEx`) is first observed, we label that window as a sensitive object. Otherwise, a call for updating the current active window (`NtUserRedrawWindow`) will occur, which allows us to mark it as a sensitive window. The outputs generated between observation of the sensitive inputs and identification of the sensitive window are all deemed sensitive. This treatment works for many Windows-based editing/viewing applications such as `Office` and `Notepad`. However, it is empirical and application-specific. A more precise approach could be tracking the file handler a window uses to retrieve the contents of the file. This needs to track Windows API calls, as such a process may not involve system calls.

Following the sensitive outputs of a process, the tracer identifies and labels new sensitive objects, such as temporary files[5], screen and clipboard. Our approach applies a set of exception rules to avoid raising the sensitivity levels of some share objects that do not contain sensitive data. Examples of such objects include "`index.dat`" that records the shortcuts of recently opened file and the registries which

[3]Such a child window usually has unique characteristics for editing and viewing purposes. In our experiments, we observed a window style parameter `0x46CF0000` associated with it in several applications.

[4]As mentioned before, most outputs of an editing/viewing application are triggered by human inputs. This allows us to establish such a correlation. However, there are exceptions. A prominent example is automatic saving, which could happen even when a document's hosting window is in the background. These outputs can be identified using application-specific exception rules. For example, many Windows-based applications create a temporary file for a document as soon as it is opened, which allows us to identify such a file and set an exception rule to keep its label consistent with that of the document.

[5]Many Windows applications tend to create temporary files for a document right after it has been opened. Some of them are used for automatically saving the contents of the document.

Name	Rule	Comments
Rule 1: editing or viewing applications	`ActiveWnd(P).sensitivity = 'high'` $\longmapsto$ `Output(P).sensitivity = 'high'&& UIO.sensitivity = 'high'`	A process `P` generates sensitive outputs and receives sensitive inputs from user input objects (UIO) when its active window is sensitive. `ActiveWnd` is a function to identify `P`'s active window and `Output` is a function to determine `P`'s outputs.
Rule 2: Web browsers	`ActivePrc(P) = 'sensitive process'` $\longmapsto$ `Output(P).sensitivity = 'high'&& UIO.sensitivity = 'high'`	The sensitive process for a web browser `P` generates sensitive outputs and receives sensitive inputs from UIOs. `ActivePrc` is a function which determines whether the process hosting the current active window is the sensitive process.

Table 1. Dependency Rules.

can be written to modify application settings.

Many Windows processes may also communicate with two system service processes, `svchost.exe` and `lsass.exe`. `svchost.exe` is a generic host process for the services running from Dynamic-link Libraries (DLLs). `lsass.exe` is a system process of the Windows security mechanisms. These processes usually provide services to applications through pipes. For simplicity, we do not trace these two processes for the time being. Instead, we label them as trusted subjects after checking the legitimacy and integrity of their executables (which includes, for example, all the DLLs `svchost.exe` hosts), and set a dependency rule which treats their replies to an application's service requests as the only outputs. Analysis of the information flows within these processes is left as our future research.

A potential problem for Rule 1 is that theoretically, active window may change between a process receiving an input from the user and the completion of the operation invoked by the input. This does not happen frequently in practice: the operations in editing/viewing applications which produce outputs (e.g., saving a file) are usually simple and finish quickly, while switch of windows usually does not happen promptly in these applications because it is driven by human interventions, which are slow.

However, Rule 1 is only meant to identify the common outputs associated with a document. Given its empirical nature and the complexity of an application's behavior, there is no guarantee that every sensitive output will be captured by the rule. Moreover, the rule may also cause false positive, labeling some public outputs as sensitive. This will happen when an output turns out to be correlated with a background window.

A tracing mechanism and a dependency rule for web browsers. Web browsers such as `Microsoft Internet Explorer` (IE) and `Mozilla Firefox` are also multithreaded applications. They provide tabs to allow the user to concurrently surf multiple websites. Tabs and browser windows can be accommodated in a single process. Different from editing/viewing applications, web browsers usually continue to serve a tab or a window even when it is not in the foreground. This is because web pages may take time to download and usually contain executable contents such as scripts. Therefore, Rule 1 cannot be directly applied to trace these applications.

To trace browsers' sensitive outputs, we developed a technique which leverages the fact that IE and `Firefox`, two most widely-used browsers, support multiple processes. Our approach uses an add-on to monitor the websites a browser is visiting. Whenever the browser tries to enter a sensitive website such as `www.citibank.com`, the add-on blocks that attempt, and instead creates a new process and directs the process to that site. We call the new process *sensitive process* and the original process *public process*. The add-on also notifies the tracer of the sensitive process's identifier for tracing its outputs to detect new sensitive objects such as temporary files and cookies. The tracer labels the UIOs as sensitive objects whenever input focus is on a window in the sensitive process. This is described by Rule 2 in Table 1. The sensitive process is not allowed to visit a website with a low sensitivity level. This is enforced by the add-on: it blocks the attempt from the sensitive process to navigate to a public URL and instead creates a tab or a window in the public process to accommodate that link.

This approach can be attacked by malicious add-ons, which stay in the same address space of a browser process and therefore could directly steal sensitive information from the process or prevent our add-on from generating a new process. These attacks can be contained by SpyShield, a technique proposed in our prior work [40], which employs a proxy to control the interactions between add-ons and their host applications.

Discussion. In this section, we present two dependency rules for editing/viewing applications and Web browsers, and describe their enforcement mechanisms. Those rules and mechanisms are developed empirically, heavily relying on application-specific knowledge. However, we argue that given the diversity of Windows applications, application-specific treatment could be an inevitable expense for fine-grained information-flow tracing in these applications without undermining their usability and changing their code. To make our approach less empirical, a potential solution could be automatic analysis of an application offline to identify its dependency rule. We may also need to dynamically in-

strument part of the application's executables, such as the mechanism for multithread communications, to implement a more precise rule. Research on this direction is left as our future work.

3.4 Sensitive Information Control

The controller works closely with the tracer to protect the sensitive information flowing out of applications, files and peripherals. It enforces the control policies of PRECIP (Section 2.3), which prevents sensitive information from flowing into untrusted subjects and the remote objects with low sensitivity levels.

Dissemination control. The controller intercepts system calls to enforce security policies. It protects a sensitive object from being accessed by an untrusted subject. Specifically, the controller blocks the calls such as `NtReadFile` from an untrusted process to read a sensitive file. It also stops untrusted code from taking a snapshot of screen through the calls such as `NtGdiStretchBlt` or directly reading from keyboard using `NtUserGetAsyncKeyState`[6] whenever these peripherals are marked as sensitive objects. For simplicity, we treat the whole screen as sensitive if a sensitive window is both visible and not minimized[7], which can be identified from the window's properties using two API calls `IsWindowVisible` and `IsIconic`. The controller also prevents an untrusted process from reading the virtual memory of a trusted process through the system call such as `NtReadVirtualMemory` whenever the latter is working on sensitive data. Another responsibility of the controller is to clean sensitive shared sources before they can be used by other parties. For example, it purges all the data inside a clipboard and the key status of a keyboard once the input focus is switched from a sensitive window to a public window.

The controller is also responsible for preventing sensitive information from being leaked to a remote object with a low sensitivity level, even through a trusted subject. To this end, it controls the network connections of a subject which are deemed dependent on a sensitive input. For example, a trusted FTP client is only allowed to have connections with sensitive domains after reading from a sensitive file. This rule, however, is hard to directly apply to web browsers: a sensitive website could host the web pages which require downloading resources such as images from many other websites. To solve this problem, we only control the websites that a browser's sensitive process is allowed to visit, not the network connections it generates during that visit, which could be highly diverse. In addition, only the sensitive process is allowed to read from sensitive files, as the public process is deemed to send the information it reads to public Internet domains.

Management of untrusted hooks. A hook is a point in Windows message-handling mechanism where an application can install a callback function to monitor and process some types of messages before they reach their target window. These messages carry information such as inputs from keyboard and mouse. Windows supports many types of hooks including keyboard, mouse, debug and others. Hooks of the same type are maintained by a hook chain, which ensures a message goes through all these hooks one by one. Windows hooks have been widely applied to enrich software's functionalities, for example, adding in new hotkeys. However, they are also intensively used by spyware to steal sensitive information. For example, many Windows-based keyloggers install keyboard hooks to intercept the user's keystrokes.

Effective management of hooks is essential to tracing and controlling message traffic. This objective, however, cannot be achieved without addressing two key challenges: first, a hook, once installed, is triggered by the kernel directly, without going through any system calls; second, a hook can be injected into the address spaces of all userland processes, which potentially allows it to touch their resources directly. We solved these problems using a framework which deploys a proxy to regulate the communication between untrusted hooks and the message-handling mechanism (Figure 2). This framework effectively protects sensitive information from untrusted hooks at a small performance overhead, as demonstrated in our experimental study (Section 5.3). We describe our approach below.

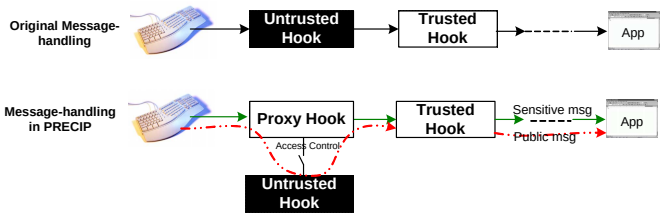

Figure 2. Management of untrusted hooks.

A hook is usually in the form of a DLL which includes a callback function. To install such a hook, a userland process must first make a system call such as `NtUserSetWindowsHookEx` to specify the location of the function and load the DLL into the message-handling mechanism. Our approach intercepts such a call from an untrusted process, blocks its attempt to hook an untrusted DLL and instead hooks a proxy DLL from another process

[6] This function reads the key status to determine whether a key is up or down at the time it is invoked and whether the key has been pressed since the previous call. After that, it resets the key status.

[7] That is, the window can be observed from screen and is not in the form of an icon on task bar.

which is trusted[8]. The untrusted DLL is loaded by a service process called *hook host*. Whenever a new message comes, the kernel first passes it to the proxy DLL within the context of the process receiving that message, and the proxy then decides whether to pass it to hook host according to the control policies. Specifically, if input focus is on a sensitive window, the proxy DLL bypasses hook host; otherwise, it sends the message to hook host that invokes the untrusted hook using the message as an input.

3.5 Integrity Protection

Without integrity protection, PRECIP is subject to a variety of attacks. For example, spyware may tamper with the trust levels of executables, the sensitivity levels of files, process images, dependency rules and security policies. Such a threat is countered by the integrity protector which offers kernel-level protection to PRECIP components.

The integrity protector regulates the system calls related to file systems (e.g., `NtWriteFile`), registry keys and other auto-start extensibility points (ASEP) [54] (e.g., `NtSetValueKey`) and processes (e.g., `NtWriteVirtualMemory`) to block unauthorized access to PRECIP components and critical system components. Specifically, it enforces the following integrity polices: (1) only the classifier can change trust-level streams and sensitivity-level streams; (2) a trusted application's files and process cannot be modified by an untrusted process; (3) dependency rules and security policies can only be accessed by a dedicated process that authenticates the user using passwords and offers an authorized party an interface protected by the tracer and the controller to change policy settings; (4) the dedicated process is also the only party that is permitted to change the ASEPs used for bootstrapping PRECIP; (5) an untrusted process is not allowed to change the registry entries used by `svchost.exe` to identify service DLLs; (6) an untrusted process cannot read or write Windows page file [14]; (7) only trusted kernel drivers are allowed to be loaded into the kernel.

The policies described above could be incomplete, missing some avenues the attacker can exploit to bypass our mechanism. However, a right policy setting will be sufficient for the integrity protector to fend off all such attacks from userland. Our design cannot handle kernel-land spyware. This threat is mitigated through regulating system calls so that only trusted kernel drivers can be loaded into the kernel. A trusted driver can be identified by comparing its hash fingerprint with those of known legitimate code, or verifying a trusted third party's signature it carries. Windows Vista and Windows XP for x64-based systems [16] also adopt a similar protection strategy.

[8]In Windows, a global hook resides in a global DLL and is mapped into individual processes independently [2]. When the message handling mechanism receives a message towards a process, it activates an instance of the hook on the stack of the recipient process. This architecture may give a malicious process a chance to tamper with the data of the proxy instance within its address space. However, it cannot directly access the instance within the address space of a trusted process. Therefore, the proxy can protect sensitive keystrokes forwarding to trusted processes. It does not protect the keystrokes towards an untrusted process, as they are deemed to be public according to PRECIP policies (Section 2.3).

4 Implementation

We implemented a prototype system to evaluate our design. Our prototype contains a kernel driver and several userland programs. The kernel driver modifies the system service dispatch table (SSDT) to wrap the system calls in `Kernel32.DLL` and `Win32.DLL`. This technique is called API hooking. Table 2 lists all the system calls hooked in our implementation.

The classifier was mainly implemented in the kernel driver. It uses the functions such as `NtWriteFile` to label the file streams for trust levels and sensitivity levels. The kernel driver maintains a 1KB array for storing four-byte identifiers of sensitive windows and an additional byte for the sensitivity levels of screen, keyboard, mouse and clipboard, each of which takes one bit. Our approach also includes a user-land program for the authorized user to modify file streams and policies. It authenticates the user using password and employs the tracer and the controller to protect itself against surveillance. User inputs are delivered to the kernel driver through the system call `NtDeviceIoControlFile`.

The tracer has a component in the kernel driver which intercepts the system calls (Table 2) related to the major output channels in Windows. These channels include shared memory, named pipes, Windows messages, files and network connections. The tracer also uses plug-ins (a toolbar in our prototype) to automatically separate the sensitive IE process from the public one: the plug-in in one process registers the IE event `BeforeNavigate2` to get the URL to be visited; it also communicates with its counterpart in the other process through named pipe and calls COM function `IWebBrowser2→Navigate()` to redirect a browser window or a tab to a website.

The controller regulates the output channels by permitting or denying relevant system calls. It also cleans the shared resources whenever input focus moves away from a sensitive windows: specifically it purges clipboard using the system call `NtUserEmptyClipboard`, and the key status of keyboard through `NtUserGetAsyncKeyState`. In addition, the controller controls Windows message traffic using the framework described in Section 3.4. To implement this framework, we used the kernel driver to block the system call for installing a hook

CATEGORY	SYSTEM CALL
File system	NtOpenFile, NtCreateFile, NtWriteFile, NtCreateSection, NtClose,
Shared memory	NtUserGetClipboardData, NtUserSetClipboardData, NtReadVirtualMemory, NtWriteVirtualMemory
Messages	NtUserPostMessage, NtUserPostThreadMessage
Named Pipe	NtCreateNamedPipeFile, NtClose
Keyboard inputs	NtUserGetKeyboardState, NtUserGetKeyState, NtUserGetAsyncKeyState
Message Hooks	NtUserSetWindowsHookEx, NtUserUnhookWindowsHookEx, NtUserCallNextHookEx
Networking	NtDeviceIoControlFile[1]
Kernel driver	NtLoadDriver
Registry keys	NtRenameKey, NtReplaceKey, NtRestoreKey, SetInformationKey, NtSetValueKey, NtDeleteValueKey
Process, thread	NtTerminateProcess, NtTerminateThread
Windows	NtUserCreateWindowEx, NtUserDestroyWindowEx, NtUserRedrawWindow
Screen	NtGDIStretchBlt, NtGDIBitBlt

Table 2. System calls hooked in the kernel driver. [1]In Windows XP, all network API calls (e.g., `send`, `recv`, `sendto`) use the device type `FILE_DEVICE_NETWORK` to communicate with the TCP/IP stack in the kernel land. The parameters of `NtDeviceIoControlFile` can be parsed to get networking information such as IP addresses, port numbers and socket actions.

(`NtUserSetWindowsHookEx`) from an untrusted process and send a request to a trusted user-land process to install a proxy hook. The trusted process further asks hook host, which is deemed untrusted, to load the original DLL. These two processes communicate with each other through named file-mapping mechanism. The kernel driver also blocks the untrusted hook's system calls such as `NtUserCallNextHookEx` which are used to transfer messages within a hook chain.

The integrity protector was completely implemented in the kernel driver. It controls the system calls in Table 2 to enforce integrity control described in Section 3.5.

5 Evaluation

In this section, we report our empirical evaluations of PRECIP using our prototype. Our purpose is to understand three major issues of our technique: (1) the accuracy of the dependency rules used in our prototype, (2) the effectiveness of our policy model in containing various types of spyware, and (3) the performance overheads of our implementation.

5.1 Dependency rules

The accuracy of dependency rules is important to the efficacy of PRECIP. Our prototype system adopts two rules, one for editing/viewing applications and the other for web browsers. The second rule is designed for monitoring a process that works entirely on the sensitive inputs. This allows us to trace all its outputs, as the BLP model does, without causing too many false positives. By comparison, the accuracy of the first rule raises more concern, as it is for identifying the sensitive outputs of a process working on both sensitive and public inputs. Therefore, we focused our attentions on the effectiveness of the first rule in the experiment. Our findings are described below.

We tested 5 common editing or viewing applications under our prototype, using Rule 1 as the default dependency rule. These applications include `Microsoft Word` 2003, `PowerPoint` 2003, `Excel` 2003, `Adobe Acrobat Reader` 8.0 and `Microsoft Notepad`. The accuracy of the rule was evaluated by its false positives that take non-sensitive outputs as sensitive ones, and false negatives that miss some sensitive outputs. In the experiment, we commanded each application to open two files: one was sensitive and the other was public. False positives of the rule were reported once the outputs of the application were found to be related to the public file when the sensitive window was in the foreground receiving inputs. False negatives were identified once we captured the outputs from the sensitive file when the public window was active.

`Microsoft Word` always maintains a single process to serve all the files being edited. In our experiment, we monitored and analyzed all the outputs of that process, including messages and operations to create and write to pipes and temporary files. The message traffic we observed was within the process, between threads and internal objects such as windows, and therefore was not of interest to us. There were three pipes connecting the process with `lsass` and `svchost`. Since we did not trace these programs, these pipes may potentially constitute a source of false negatives. The process also wrote to an existing file (`index.dat`) and created temporary files and several `.LNK` shortcut files. This all happened when the process opened the file to be edited, created its editing window and put the window in the foreground. Our tracer captured these files and associated them with the file being edited in the active window. We also found that all the temporary files created when an editing window was in the foreground were automatically removed after that window was closed. This

indicates that these files were only related to the active window, not to the editing window in the background. Other outputs from Word were purely driven by human inputs, such as "save" and "cut and paste". Our prototype successfully captured all those outputs generated by the sensitive window.

Like Word, PowerPoint, Excel and Acrobat Reader all run in single-process mode. PowerPoint and Excel created .LNK shortcuts for files when these files were just opened and their application windows were active. The shortcuts related to sensitive files were correctly identified by our prototype. PowerPoint and Acrobat Reader also had pipes with lsass, which could cause false negatives. Most other outputs of these applications were directly driven by the human inputs and well modeled by Rule 1. Notepad generated a new process to serve an editing window. The process did not produce any outputs when its editing window was in the background. This allowed our prototype to catch all the outputs related to the sensitive file.

This study is still preliminary: more conclusive results should come out of evaluation of our implementation under practical settings. For example, we can distribute it to a number of work computers to study its false positive/negative rates during day-to-day operations. Another possibility is to evaluate the implementation using a comprehensive test suite which automatically checks most of functionality of an application[9]. These measures will definitely help us better understand the accuracy of our dependency rule. In addition, we labeled the files in the experiment manually. Future research will study the accuracy of the automatic classification techniques described in Section 3.2.

5.2 Effectiveness

We tested the PRECIP mechanism against 10 strains of spyware, including keyloggers, screen grabbers and file stealers. The results of this study are summarized in Table 3 and elaborated below.

Keyloggers. Most in-the-wild keyloggers for Windows use hooks to intercept keystrokes. Specifically, they need to load a DLL into the message-handling mechanism so as to access the messages transporting keystrokes to application windows. In our experiments, we tested 3 keyloggers of this type: Spyware.KidLogger [50], Home KeyLogger v1.60 [8] and RunHook [19]. Spyware.KidLogger is a famous spyware, which has been rated as high risk impact by Symantec [50]. This spyware hooks the message queues for both keyboard (WH_KEYBOARD) and mouse (WH_MOUSE), and maintains a record for every application. Home KeyLogger hooks two types of messages: WH_GETMESSAGE and WH_KEYBOARD. RunHook works in a similar fashion as these two keyloggers, but only intercepts keyboard messages. Our prototype uses a proxy to regulate these untrusted hooks' access to message traffic. This turned out to be very effective: in our experiment, we used Word to edit a sensitive file and a public file in the presence of these keyloggers; from their log files, we only found keystroke records for the public file.

Besides keyboard hooking, there are other ways to log keystrokes. Pranay Kanwar described a keylogging technique using GetAsyncKeyState [34]. A thread periodically polling the keyboard with this function can capture the keys just being pressed. Kanwar also provided part of the source code for this technique. Another hookless approach takes advantage of the Win32 call AttachThreadInput to synchronize the input processing of spyware with that of the thread hosting the active window, and then calls GetKeyboardState to acquire the window's keystroke inputs [27]. Both these techniques require making system calls (NtUserGetAysncKeyState, NtUserGetKeyState, NtUserGetKeyboardState) to read from keyboard, which can be blocked when keyboard is sensitive. In our experiment, we implemented both approaches and ran them against our prototype. Although they could successfully record keys when we were editing a public file, none of them were found to be able to do that when input focus was on a sensitive window.

Screen grabbers. We tested our prototype against 3 real screen grabbers: GhostlyEye v1.0 [7], Any Capture v3.12 [4] and OleanSoft Hidden Recorder v1.9 [12]. All of them are capable of periodically dumping the image from the screen to a log file. OleanSoft even disguises the file as a DLL. To capture the screen, these screen grabbers made system calls NtGDIStretchBlt and NtGDIBitBlt, which was intercepted by the kernel driver. The control policies of our prototype forbid an untrusted process to read from screen whenever a sensitive window is both visible and not minimized, even when the window is not active. In the experiment, we edited a sensitive file and then closed it in the presence of the spyware. From the images exported by these grabbers, we found that none of them contained the editing window for the sensitive file.

File stealers. We also evaluated our technique using 2 file stealing tools: Backdoor.Sub7 [15] and Cerberus FTP Server v2.45 [5]. Sub7 is a famous backdoor which contains a server and a client. The server is running on the victim's system and the client provides a graphic user interface for controlling the server. Through the client, one can run a keylogger on an infected system or directly download files from the system. The keylogger used by Sub7 is based on keyboard hooking, which can be easily contained by our

[9] We are not aware of any publically-available test suite that can be used for this purpose. This is the major reason that prevents us from conducting a more comprehensive test of the rule.

	Name	Type	Control Actions
1	KidLogger [50]	Key Logger	bypass the hook host.
2	Home KeyLogger [8]	Key Logger	bypass the hook host.
3	RunHook [19]	Key Logger	bypass the hook host.
4	Synthesized-1[27]	Key Logger	block two system calls: `NtUserGetKeyboardState` and `NtUserGetKeyState`.
5	Synthesized-2[34]	Key Logger	block one system call: `NtUserGetAsyncKeyState`.
6	GhostlyEye[7]	Screen Grabber	block one system call: `NtGDIStretchBlt`
7	Any Capture[4]	Screen Grabber	block two system calls: `NtGDIStretchBlt` and `NtGDIBitBlt`
8	Hidden Recorder[12]	Screen Grabber	block one system call: `NtGDIBitBlt`.
9	Sub7[15]	File Stealer	untrusted process does not allow to open sensitive files.
10	Cerberus[5]	Lightweight ftpd	untrusted process does not allow to open sensitive files.

Table 3. Effectiveness Evaluation of our PRECIP Prototype.

Benchmark	Baseline	with PRECIP	Overhead
Office XP SP2	784 s	838 s	6.89%
Photoshop 7.0.1	647 s	675 s	4.33%
Mozilla 1.4	1122 s	1265 s	12.75%

Table 4. Overhead of the Kernel Driver.

prototype. Therefore, we are more interested in its file stealing functionality. Cerberus is essentially a lightweight FTP server which allows an FTP client to access the file system of the host it is running on. Choosing Cerberus is motivated by the observation that one can use such a lightweight server to steal files remotely. In our experiment, we installed both of these programs in a system protected by our prototype and ran their remote clients to download files from the system. In both cases, we successfully downloaded files with low sensitive levels but could not touch sensitive files. This is because our kernel driver prevents untrusted processes from reading sensitive files through system calls such as `NtOpenFile` and `NtCreateFile`.

5.3 Performance

We evaluated the performance of the kernel driver and the hooking management mechanism, two major components of our prototype. The experiments were conducted in a VMware workstation hosted by a desktop with Intel Pentium 2.53GHz CPU. The virtual machine has 780MB memory and 16GB disk space at its disposal. Its operating system is Windows XP with Service Pack 2. The experimental results are described below.

Performance of the kernel driver. We used WorldBench 5.0 [13] to measure the performance impacts of our kernel driver. WorldBench is an industry-standard benchmarking application which has been widely used to measure the performance of personal computers. It automatically executes several common applications using artificial tasks to determine a system's performance. In our experiment, we ran WorldBench in the virtual machine to get baseline performance, and then ran it again after installing the kernel driver. Table 4 presents the results, which are averaged over three tests.

From the table, we can see the overhead of the kernel driver is small, always below 13%. In particular, it only brought 6.89% performance impact to Microsoft office, which it is meant to protect.

Performance of the hooking management mechanism. Legitimate freeware can also install a hook to the message-handling mechanism to provide services such as hotkeys. The performance of such a hook will be affected by our hooking management mechanism. In addition, a hook slows down the performance of the whole system. A question is whether our framework makes the situation much worse. To understand how serious the problem is, we empirically studied the performance of our mechanism and report our findings here.

In the experiment, we first installed a test hook (the DLL of KidLogger), ran a program called *keystroke generator* to produce keystrokes, and delivered them to another program called *receiver*. Keystroke generator and receiver coordinated to measure the delay caused by delivering keystroke messages, which was used as a baseline. Then, we enabled our prototype that hooked a proxy DLL to the message-handling mechanism and employed hook host to load the test hook. Running the keystroke generator again, we measured the delay and compared it with the baseline. In the experiment, 1000 keystrokes were generated and delivered in both settings. The experiment results are the average delays for transiting these keystrokes. The baseline is 691.015 microseconds and the average delay caused by our prototype is 784.809 microseconds. This gives a performance overhead of 13.57%.

6 Discussion

A limitation of the PRECIP model is that it has only two sensitivity levels and does not consider compartmentalization of information. These issues are left out in the current design for the simplicity of the model. In addition, it is also desired to have a policy language that translates the high-level policies specified by PRECIP to the policies enforceable by a general policy enforcer for an operating system. These limitations are expected to be addressed in our future research.

The dependency rules we present in the paper are empirical. When applying them to a real application, it is likely that they miss some sensitive outputs. The problem is aggravated by the fact that even the applications from a renowned software vendor may contain hidden channels. A famous example is Sony BMG copy protection scandal [17]. However, in most cases, legitimate applications themselves do not steal the user's confidential data and therefore failing to monitor some of their outputs will not automatically cause information leakage, though this gives a knowledgeable adversary chances to do so [10]. On the other hand, we can always patch our dependency rules to fix such holes as soon as we know them. Essentially, our intention to retrofit existing systems decides that we have to step back from the ambition of achieving perfect security. Pragmatically, we just want to raise the bar through controlling the major channels the adversary uses to compromise system confidentiality.

The proposed dependency rules are also limited in their applicable scope which only includes editing/viewing applications and web browsers. An open research question is how to find efficient, accurate and general policies for other types of applications. In general, tracing a multitasked process is hard because of its nondeterministic behaviors. We are developing the technique for automatically analyzing a commodity application offline to generate its dependency rule that can be enforced online through dynamic instrumentation.

The controller we implemented regulates system calls and the message-handling mechanism. There are other channels through which sensitive information can be leaked out. An example is the channel between an application and its DLLs. To control such a channel, we may separate an untrusted DLL from its host application and use a proxy to manage the information flows between them. The controller is also limited by its capability to clean shared resources. Our prototype only cleans clipboard and keyboard, which is far from sufficient: previous research shows that sensitive data could be scattered in the operating system, being left in the places such as stack and heap [24]. Therefore, an important question is how to thoroughly clean a process once it finishes a task involving sensitive data. An existing solution to this problem requires modifying OS source code, which may not be suitable for retrofitting a commercial system [25]. We plan to seek other solutions in the follow-up research.

Although we implemented our prototype under Windows XP, our model can also be applied to Linux using the techniques such as LSM hooking [55]. Actually, identifying dependency rules for Linux applications should be easier, as many of them are not multithreaded. Windows XP for x64-based systems and Windows Vista include a mechanism for kernel patch protection that disallows system-call interception through unauthorized modification of kernel resources such as hooking system-call dispatch table. However, Microsoft does provide alternatives to kernel patching, which allow a third-party program to monitor and control network traffic, operations on file systems and registry entries, and others [16]. This could make it possible to enforce PRECIP policies in userland. Further investigation of this problem is left as our future research.

The PRECIP mechanism does not interfere with the operations of other malware defense programs such as Symantec AntiVirus. We can label these programs as trusted and allow them to directly hook the message-handling mechanism or load the kernel driver, which are necessary for accomplishing their missions.

[10] For example, Sony BMG's extended copy protection (XCP) does not steal the user's password. However, it contains security holes which might be exploited by other malware [17].

7 Related Work

One of the earliest security policy model aiming at preventing sensitive information to leak is the Bell-LaPadula model [21]. The BLP model suffers from several limitations. First, it is unsuitable for modeling a multitasked process which works concurrently on public and sensitive information. Application of the model to a practical system usually requires declaring a large number of subjects to be trusted. This does not work well in a modern operating system such as Windows in which almost all the programs are multitasked. Second, the BLP model does not describe shared resources, in particular, the user input objects which are essential to the confidentiality assurance of a modern OS. The PRECIP model is designed to solve these problems, towards the goal of providing practical confidentiality protection for commercial systems.

There is a large body of work on precisely defining information flow properties. One prominent example is *noninterference* property which requires that secret information not affect publicly observable behavior of a system [30]. A system with this property does not leak out any sensitive information through even covert channels. However, such a property is shown to be too restricted for many practical systems [49, 38]. Language-based information-flow security [47] seeks to develop programming-language techniques for specifying and enforcing information flow policies. These techniques are meant to be used for writing new programs, while PRECIP is designed for retrofitting the existing applications and systems without access to their source code. PRECIP takes the definition of dependency relation as an input to the model, and focuses on tracing information flows using that notion. This gives it sufficient flexibility to be applied to many practical systems. Language-based information-flow techniques can be complementary

to PRECIP in that they can be used to more precisely determine dependency relation within one program. Recently, effort has been made to dynamically trace the information flow in an executable using binary instrumentation [51, 46] or virtual machines [29, 43]. These approaches are generally too slow to use online. Actually, some of them are developed for offline analysis of vulnerable programs or malware [29, 43].

Some prototype operating systems have built-in information-flow security. For example, Asbestos [28] provides a kernel-enforced labeling mechanism; IX [42] modifies UNIX kernel to implement multilevel security. PRECIP differs from them in its objective to retrofit existing systems. In addition, PRECIP also deals with some practical issues unaddressed in these systems. A prominent example is the policy to identify the sensitivity level of a user input object such as keyboard.

Sandboxing techniques have also been applied to achieve multilevel security. For example, WindowBox [20] virtually divides a Windows workstation into multiple desktops, each of which is sealed off from the others. Another example is the NetTop project [11] which uses VMware for multilevel security. These approaches have two major limitations. First, they tend to introduce heavy performance overheads, as a result of using virtual machines and installing the same software within different compartments. Second, they require users to correctly partition their activities to different security levels, whereas many activities may span multiple levels, e.g., reading and handling emails. By comparison, PRECIP incurs small overheads and is capable of allowing the users to work in one unified environment: for example, our model allows one subject to handle data with different sensitivity levels.

Wrappers is a toolkit that wraps commodity software at the system-call level to enhance the software's security and reliability [36]. It can serve as a platform for partially enforcing the new policy model we propose here, in particular when sensitive information flows are observable through system calls. However, the PRECIP model may not be fully enforceable using Wrappers, as some information flows do not go through system calls: an example is the message-handling mechanism which allows the OS kernel to directly invoke a user-defined callback function. Actually, in our research, we developed our own kernel module to intercept system calls because the current prototype of Wrappers does not support Windows XP [36, 45].

Flume is a model for decentralized information flow control, which works on the granularity of processes [37]. The model can be used to protect both integrity and confidentiality, and therefore is more general than PRECIP. However, it does not describe user input objects and multithreaded subjects, two long-standing problems for process-level information flow models. In contrast, PRECIP is designed to address these issues. Another difference is that Flume requires modifying the source code of operating systems and applications, while PRECIP does not.

Many spyware detection techniques have been proposed recently. Examples include Panorama [57], Siren [23], NetSpy [52] and others [29]. Different from these approaches, the focus of PRECIP is containment, which offers another layer of confidentiality protection even after spyware evades detection. Existing containment techniques are limited to protecting certain type of information such as passwords. Bump in the Ether [41] offers a mechanism to bypass common avenues of attacks on keystroke inputs through a trusted tunnel implemented using a mobile device. SpyBlock [32] evades the surveillance of the keyloggers inside a virtual machine by directly injecting users' passwords into the network traffic intercepted by the host. These approaches are ineffective against other types of spyware such as screen grabbers and file stealers. In addition, they need either additional hardware (mobile device) or heavyweight software (a virtual machine). PRECIP is designed to offer general protection against multiple types of spyware and has no special requirements for hardware and software settings.

8 Conclusions and Future Work

In this paper, we propose PRECIP, a new confidentiality model, as a first step towards practical and retrofittable confidential information protection. This model is designed to be used in practical systems, offering an efficient online protection against spyware surveillance without touching the source code of these systems. To this end, PRECIP addresses several important practical issues which have not been modeled in previous research, including models for human input objects, shared objects and multitasked subjects. We applied the PRECIP model to Windows XP to protect the commercial applications for editing or viewing sensitive documents and browsing sensitive websites, and evaluated its efficacy using our prototype. Future research includes extending the model to describe the sensitivity levels in lattice and developing general, accurate and efficient tracing techniques.

9 Acknowledgements

The authors thank Xuxian Jiang, the anonymous reviewers their comments on the draft of the paper. This work was supported in part by the National Science Foundation the Cyber Trust program under Grant No. CNS-0716292.

References

[1] State of Spyware Q2 2006, Consumer Report. http://www.webroot.com/resources/stateofspyware/excerpt.html.

[2] Hooks and DLLs. http://www.flounder.com/hooks.htm, as of December 2007.

[3] About keyboard input, as of May 2007. http://msdn2.microsoft.com/en-us/library/ms646267.aspx.

[4] Any capture v3.12, as of May 2007. http://www.any-capture.com/.

[5] Cerberus ftp server v2.45, as of May 2007. http://www.cerberusftp.com/.

[6] Description of office features, as of May 2007. http://support.microsoft.com/kb/822924.

[7] Ghostlyeye - the surveillance screen grabber v1.0, as of May 2007. http://www.reg.net/product.asp?ID=14992.

[8] Home keylogger v1.60, as of May 2007. http://www.cs.stthomas.edu/faculty/resmith/r/mls/index.html.

[9] How ntfs works, as of May 2007. http://technet2.microsoft.com/windowsserver/en/library/8cc5891d-bf8e-4164-862d-dac5418c59481033.mspx?mfr=true.

[10] Important aspects of password and encryption protection, as of May 2007. http://office.microsoft.com/en-us/ork2003/HA011403111033.aspx.

[11] A network on your desktop, as of May 2007. http://www.vmware.com/pdf/TechTrendNotes.pdf.

[12] Oleansoft hidden recorder v1.9, as of May 2007. http://www.oleansoft.com/hiddenrecorder.htm.

[13] Pc world worldbench 5.0, as of May 2007. http://www.pcworld.com/article/id,116888-page,1/article.html.

[14] Ram, virtual memory, pagefile and all that stuff, as of May 2007. http://support.microsoft.com/kb/555223.

[15] Subseven 2.2.0, as of May 2007. http://hackpr.net/~sub7/main.shtml.

[16] An Introduction to Kernel Patch Protection. http://blogs.msdn.com/windowsvistasecurity/archive/2006/08/11/695993.aspx, as of November 2007.

[17] 2005 Sony BMG CD copy protection scandal. http://en.wikipedia.org/wiki/2005_Sony_BMG_CD_copy_protection_scandal, as of September 2007.

[18] W. Ames. Understanding spyware: Risk and response. *IT Professional*, 6(5):25–29, 2004.

[19] Arkon. Tutorial - keyboard hook, 29-May 2003. http://www.ragestorm.net/tutorial?id=10.

[20] D. Balfanz and D. Simon. Windowbox: A simple security model for the connected desktop. In *Proceedings of 4th USENIX Windows Systems Symposium*, August 2000.

[21] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. MTR-2997, (ESD-TR-75-306), available as NTIS AD-A023 588, MITRE Corporation, 1976.

[22] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of USENIX Security Symposium 2003*, August 2003.

[23] K. Borders, X. Zhao, and A. Prakash. Siren: Catching evasive malware (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 78–85, 2006.

[24] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[25] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[26] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Usenix Security Symposium*, pages 63–78, Jan 1998.

[27] dimport. How to write a hookless key logger for windows, 21-June 2003. http://www.osix.net/modules/article/index.php?id=162.

[28] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, 2005.

[29] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *To appear in the 2007 USENIX Annual Technical Conference*, June 17-22 2007.

[30] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, Apr. 1982.

[31] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 257–268, 2006.

[32] C. Jackson, D. Boneh, and J. C. Mitchell. Stronger password authentication using virtual machines. In submission, Stanford University, 2006.

[33] A. K. Jones and R. J. Lipton. The enforcement of security policies for computation. In *SOSP*, pages 197–206, 1975.

[34] P. Kanwar. The art of key logging - implementation and detection on windows platform. http://warl0ck.metaeye.org/arts/art_keylog.txt.

[35] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proceedings of 15th USENIX Security Symposium*, August 2006.

[36] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 11–11, Berkeley, CA, USA, 2000. USENIX Association.

[37] M. N. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SOSP*, pages 321–334, 2007.

[38] C. E. Landwehr. Formal models for computer security. *ACM Comput. Surv.*, 13(3):247–278, 1981.

[39] L. J. LaPadula and D. E. Bell. Secure computer systems: A mathematical model. *Journal of Computer Security*, 4:239–263, 1996.

[40] Z. Li, X. Wang, and J. Y. Choi. Spyshield: Preserving privacy from spy add-ons. In *RAID*, pages 296–316, 2007.

[41] J. M. McCune, A. Perrig, and M. K. Reiter. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 185–198, June 2006.

[42] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software - Practice and Experience*, 22(8):673–694, 1992.

[43] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, May 2007.

[44] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[45] N. Provos. Systrace - Interactive Policy Generation for System Calls. `http://www.citi.umich.edu/u/provos/systrace/`, as of September 2007.

[46] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148, 2006.

[47] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[48] W. Shi, H.-H. Lee, G. Gu, L. Falk, T. Mudge, and M. Ghosh. InfoShield: A security architecture for protecting information usage in memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA'06)*, March 2006.

[49] R. Smith. Introduction to multilevel security, as of May 2007. `http://www.kmint21.com/keylogger/`.

[50] S. K. Specifications. `http://www.symantec.com/security_response/writeup.jsp?docid=2006-020913-4035-99`, as of Nov. 2006.

[51] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO*, pages 243–254. IEEE Computer Society, 2004.

[52] H. Wang, S. Jha, and V. Ganapathy. NetSpy: Automatic Generation of Spyware Signatures for NIDS. In *Proceedings of ACSAC*, 2006.

[53] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: black-box exploit detection and signature generation. In *ACM Conference on Computer and Communications Security*, pages 37–46, 2006.

[54] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *USENIX LISA 2004*, 2004.

[55] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, pages 17–31, 2002.

[56] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[57] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[58] S. Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.